FPT UNIVERSITY, QUY NHON CITY
FACULTY OF ARTIFICIAL INTELLIGENT

FPT UNIVERSITY

REINFORCEMENT LEARNING (REL301m)

Assignment

“Cutting stock problem”

Instructor(s):	Nguyễn An Khương
Report team:	Group 02
Students:	Nguyễn Lê Quốc Việt - QE170144 (Leader) Lê Quốc Việt - SE173577 Lê Văn Chương - QE170039 Trần Hữu Hoàng - QE170011 Nguyễn Ngọc Phúc - QE170087

QUY NHON CITY, MARCH 2025

Member list & Workload

No.	Full Name	ID	Problems	% done
1	Lê Quốc Việt	SE173577	1. Implement the Reinforcement Learning Approach with with PPO (Proximal Policy Optimization) and DQN (Deep Q Network) 2. Algorithm research (Theory behind RL, DQN and PPO)	100%
2	Lê Văn Chương	QE170039	1. Implement the Reinforcement Learning Approach with Q-Learning	100%
3	Nguyễn Lê Quốc Việt	QE170144 (Leader)	1. Implement the Reinforcement Learning Approach with Q-Learning	100%
4	Nguyễn Ngọc Phúc	QE170087	1. Implement the Heuristic Algorithm with First Fit, Best Fit, Combination Algorithm	100%
5	Trần Hữu Hoàng	QE170011	1. Implement the Reinforcement DQN (Deep Q Network) 2. Algorithm research (Theory behind DQN)	100%

1. Introduction

1.1. The Cutting-Stock Problem

The Cutting-Stock Problem (CSP) is a well-established NP-hard optimization problem with significant applications in real-world industries. It can be defined as follows:

> Given a set of large stock materials with fixed dimensions, and a set of required smaller items, the objective is to determine the optimal cutting patterns that minimize waste and the number of stock materials used, while ensuring that all demands are satisfied.

This problem is particularly relevant in manufacturing sectors such as paper, textiles, and metal production, where large sheets, rods, or rolls of material need to be cut into smaller sizes to fulfill customer orders in an efficient manner.
The Cutting-Stock Problem is typically formulated as an Integer Linear Programming (ILP) problem. Several solution methods have been proposed, including dynamic programming, branch-and-bound, and column generation techniques, each offering trade-offs between computational efficiency and solution optimality.
For the purposes of this study, we focus on a variation known as the **one dimensional cutting stock problem**, where the items to be cut are assumed to be one-dimensional lengths (such as lengths of wood or metal bars) that must be cut from one-dimensional stock materials, without rotation.

1.2. What is Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning where an agent learns to make decisions by interacting with an environment. Unlike supervised learning, where a model is trained on labeled data, RL involves learning through trial and error, using feedback from the environment in the form of rewards or punishments. The goal is to maximize the cumulative reward over time.
Here's a breakdown of key concepts in **Reinforcement Learning**:

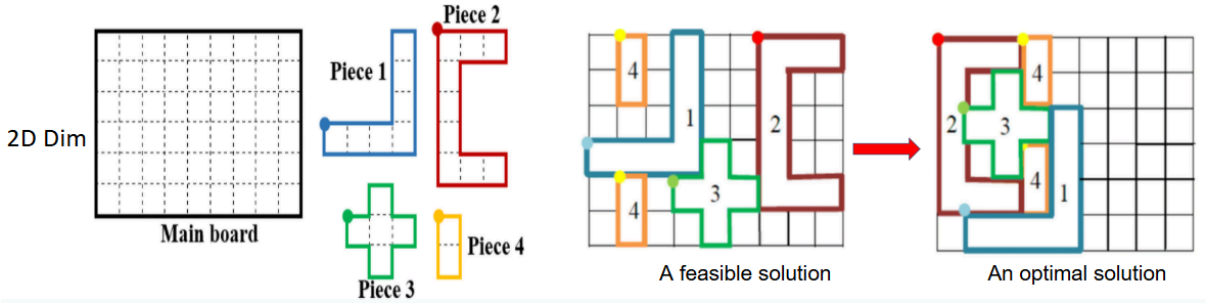
- **Agent**: The learner or decision maker, which interacts with the environment.
- **Environment**: The external system with which the agent interacts. The environment defines the rules, states, and actions available to the agent.
- **State (s)**: A representation of the environment at a given time. The state contains all the information the agent needs to make a decision.
- **Action (a)**: The choices the agent can make at each state. These actions affect the environment and change the state.
- **Reward (r)**: A scalar value that the agent receives after taking an action in a state. Rewards guide the agent toward better actions.
- **Policy (π)**: A strategy or rule that defines the agent's behavior. It maps states to actions, indicating the action the agent will take in a given state.

1.3 Cutting Stock Problem In Reinforcement Learning

Traditionally, the **Cutting Stock Problem (CSP)** has been solved using methods like **Linear Programming** and **Heuristic Techniques**. However, these approaches face challenges when dealing with **large state spaces**, **complex structures**, and **high uncertainty**, especially in larger-scale problems or those with multiple dimensions.
Reinforcement Learning (RL) has emerged as a promising solution for CSP due to its ability to **learn from experience** and improve over time. Unlike traditional methods, RL can handle **large and complex state spaces** effectively and adapt to various problem scales.

Reinforcement Learning emerges as a new approach to solving CSP with the following advantages:

- Ability to learn from experience and improve over time.
- Effectively handle problems with large and complex state spaces.
- Can be applied to complex variants of CSP (2D, 3D).



2. Acknowledge

Our group expresses deep gratitude to our instructor, Mr. Nguyễn An Khương for his guidance and valuable feedback throughout the creation process of this big assignment.

3. Algorithms

3.1. Heuristic Algorithms

3.1.1 What are Heuristic Algorithms?

Heuristic algorithms are problem-solving methods that rely on **rules of thumb**, **intuition**, and practical approaches to find **approximate solutions** for complex problems. Rather than searching for an exact solution, heuristic algorithms aim to generate a solution that is good enough within a reasonable timeframe.

These algorithms are particularly valuable when the **exact optimal solution** is difficult to compute due to the **large problem space** or when finding such a solution is **computationally expensive**. In many real-world applications, especially in fields like optimization, scheduling, or routing, obtaining the perfect solution might be impractical due to time limitations or the complexity of the problem.

The primary goal of a heuristic is not to find the absolute best solution, but rather to arrive at a **satisfactory solution** that meets the problem's requirements. These methods can offer a practical alternative by providing feasible answers in a shorter amount of time, making them useful for problems where an optimal solution is unnecessary or too costly to compute.

Heuristic algorithms are widely used in areas like artificial intelligence, operations research, and machine learning, where problems often involve large data sets or variables that make exact solutions infeasible. Examples of heuristic methods include **greedy algorithms**, **simulated annealing**, and **genetic algorithms**, each tailored to different types of problems.

3.1.2 Heuristics in Cutting Stock Problem (CSP)

In the **Cutting Stock Problem (CSP)**, **heuristics** play a crucial role in finding feasible cutting options, minimizing material waste, and saving costs. Since the CSP involves large-scale optimization with numerous possible cutting patterns, heuristics provide a practical solution by quickly exploring a manageable subset of possible options.

Heuristic algorithms often incorporate specific **constraints**, such as the **limited number of cuts** or the use of a **guillotine cutting style** (which assumes that each cut is straight and extends across the entire material). These constraints help to **minimize the search space**, making the problem more tractable and speeding up the solution process.

Although heuristics do not guarantee a **global optimal solution**, they are frequently effective in practice, especially when decisions need to be made quickly. They allow businesses to **optimize resources** and minimize waste efficiently, even if the solution found is not the absolute best. This makes heuristics a powerful tool for real-time applications where computational resources or time constraints may limit the ability to find the optimal solution.

3.2 Applying Heuristic Algorithms in CSP

In the **Cutting Stock Problem (CSP)**, several heuristic methods are employed to find efficient solutions for minimizing material waste and optimizing resource use. Three popular heuristics used in CSP include:

First-Fit Algorithm:

- The First-Fit algorithm places each item in the **first available location** that can accommodate it without violating any constraints. This method is **simple, fast**, and often used as a baseline for comparison.
- While it is a quick and straightforward approach, the First-Fit algorithm **does not guarantee the global optimal solution**, as it does not explore all possible placements and can miss better alternatives.

Best-Fit Algorithm:

- The Best-Fit algorithm places each item in the **location with the smallest available space** that is still sufficient to accommodate it. This helps to **optimize space utilization** and **reduce material waste**.
- Although this method is more effective than First-Fit in many cases, it requires more checks and calculations. Like the First-Fit algorithm, it may still miss the **optimal solution** in certain situations.

Greedy Algorithm

- The Greedy algorithm always selects the **best cutting option** at the current time, in the hope of achieving the global optimum by making local decisions that seem best at each step.
- This method is **fast**, **easy to implement**, and **effective in practice**. However, it is prone to getting stuck in **local optima**, meaning it might not find the best solution across all possible configurations.

3.2.1 First-Fit

The **First-Fit Algorithm** is a simple and efficient heuristic method designed to solve **2D Cutting-Stock Problems**, where the objective is to pack rectangular items onto stock sheets while minimizing material waste. The algorithm focuses on placing each product on the first stock sheet that has enough available space to fit it, ensuring a balance between **simplicity** and **speed**. While this method does not guarantee an optimal solution, it is often computationally feasible and works well in many real-world scenarios.

Algorithm Steps:

- **Product Selection**: The products are considered and placed in the order they are presented in the input.
- **Finding the First Available Stock**: For each product, the algorithm checks each available stock sheet in sequence to determine whether the product fits, considering both the remaining width and height of the stock sheet.

- **Placing the Product**: If the product fits within the available space on an existing stock sheet, it is placed there. If no suitable stock sheet is found, a new stock sheet is introduced, and the product is placed on it.
- **Repetition**: The process is repeated for all products until they are all placed onto stock sheets.

Notations:

- Let **n** be the number of products.
- Let **m** be the number of stock sheets used.
- Let **Wi** and **Hi** be the width and height of the i-th product.
- Let **Wstock_j** and **Hstock_j** be the width and height of the j-th stock sheet.

Objective:

- The goal of the **First-Fit Algorithm** is to **minimize the number of stock sheets used** while ensuring that each product is placed within the dimensions of the stock sheets available.

Placement Process:

- For each product, find the first stock sheet **j** such that **Wi ≤ remaining width** and **Hi ≤ remaining height**.
- If no stock sheet can accommodate the product, a new stock sheet is created, and the total number of stock sheets **m** is incremented.

Pseudo Code:

```
Function First_fit_2DCSP:
    push one of each stock_type into the stock_list.
    for stock_i from the largest to smallest area in stock_list:
        for prod_j from the largest to smallest area in product_list:
            if prod_j can fit into stock_i:
                place prod_j into stock_i
            else:
                let "stockType" be the first stock_type that fits prod_j
                create a new stock of type "stockType"
                place prod_j into the new stock
            if the textbfdemand for prod_j is textbfmet:
                remove prod_j from the item list
```

Time Complexity:

The time complexity of the **First-Fit Algorithm** for 2D Cutting-Stock is **O(n · m)**, where:

- **n** is the number of products to be placed,
- **m** is the number of stock sheets used.

❑ Cutting Data

Stock Sheet Information

Stock ID	Length (cm)	Width (cm)
1	120	60
2	100	50
3	90	40
4	80	30

Demand Pieces

Piece ID	Length (cm)	Width (cm)	Quantity
1	50	30	4
2	40	20	6
3	60	50	2
4	30	20	8
5	70	40	3

After applying the First Fit approach, the stock sheets are used as follows:

Summary Table

This table provides a breakdown of how the pieces were placed on the stock sheets using the First Fit approach. Each row represents a piece that was successfully placed, detailing the stock sheet it was placed on, the dimensions of the piece, and its position on the stock sheet.

Summary of First Fit Cutting:

Stock ID	Piece ID	Dimensions	Position
1	1	50x 30	(0,0)
1	1	50x 30	(50,0)
1	1	50x 30	(0,30)
1	1	50x 30	(50,30)
2	2	40x 20	(0,0)
2	2	40x 20	(40,0)
2	2	40x 20	(0,20)
2	2	40x 20	(40,20)
3	2	40x 20	(0,0)
3	2	40x 20	(40,0)
3	4	30x 20	(0,20)
3	4	30x 20	(30,20)
3	4	30x 20	(60,20)
4	4	30x 20	(0,0)
4	4	30x 20	(30,0)

Waste Summary

Total Area = Stock Length × Stock Width

Used Area = Sum of all placed pieces' areas

Waste Area = Total Area - Used Area

Waste Summary:

Stock ID	Used Area	Waste Area
1	6000	1200
2	3200	1800
3	3400	200
4	1200	1200

Total Waste Area: 4400 square units

Reward System

- To assess the efficiency of the cutting strategy, a reward system is introduced:
  - Penalty of **-10 points** for each piece that could not be placed.
  - The final reward score helps evaluate the effectiveness of the approach.

```
Cannot place piece 3
Cannot place piece 3
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Cannot place piece 5
Cannot place piece 5
Cannot place piece 5

Total Reward: -80
```

3.2.2 Best-Fit

The **Best-Fit Algorithm** is a widely used heuristic for solving **2D Cutting-Stock Problems**, aimed at **minimizing material waste** when placing rectangular items (or products) onto stock sheets. Unlike the **First-Fit Algorithm**, which simply places each item on the first available stock sheet, the **Best-Fit Algorithm** strives to place each product on the stock sheet that leaves the **least amount of unused area**, which results in better utilization of available material.

Algorithm Steps:

- **Product Selection**: Products are placed one by one, in the order in which they are provided.
- **Finding the Best Fit**: For each product, the algorithm checks each available stock sheet (in the order they are considered) to determine which one, after placing the product, leaves the **smallest unused area**. This helps to optimize space and minimize waste.
- **Placing the Product**: If a product fits on an existing stock sheet, it is placed at the position that leaves the smallest leftover area. If no stock sheet can accommodate the product, a new stock sheet is introduced, and the product is placed on it.
- **Repetition**: This process continues until all products are placed onto stock sheets.

Notations:

- Let **n** be the number of products.
- Let **m** be the number of stock sheets used.
- Let **Wi** and **Hi** be the width and height of the i-th product.
- Let **Wstock_j** and **Hstock_j** be the width and height of the j-th stock sheet.

Objective:

- The main goal of the **Best-Fit Algorithm** is to **minimize the remaining unused area** on stock sheets, thus reducing material waste. This approach ensures that the available stock sheets are utilized as efficiently as possible.

Placement Process:

- For each product, find the stock sheet **j** such that the **remaining area after placing the product** is minimized, i.e., minimize the remaining area where **Wi ≤ remaining width** and **Hi ≤ remaining height**.
- If no stock sheet can accommodate the product, introduce a new stock sheet and increment **m** (i.e., **m = m + 1**).

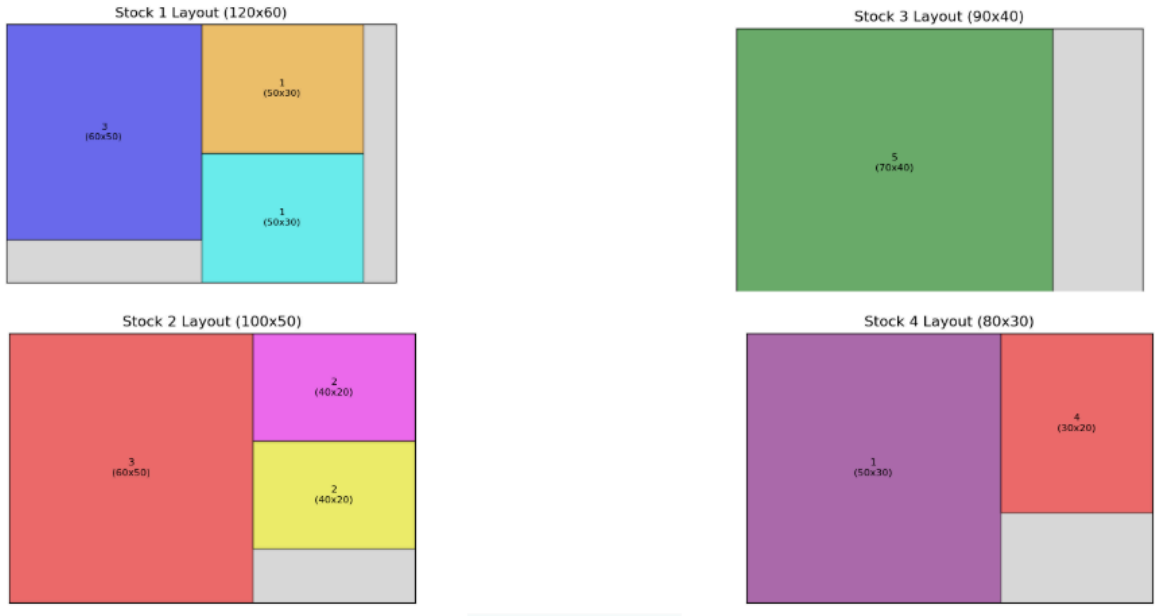
Time Complexity:

- The time complexity of the **Best-Fit Algorithm** for 2D Cutting-Stock is **O(n · m)**, where:
- **n** is the number of products,
- **m** is the number of stock sheets used.

Enhancement Opportunities:
The Best-Fit Algorithm can also be enhanced by introducing additional strategies, such as:

- **Rotating products** to utilize the stock sheet space better,
- **Prioritizing smaller products** to ensure that larger stock sheets are used more effectively for bigger products.

After applying the Best Fit approach, the stock sheets are used as follows:



Summary Table
This table provides a breakdown of how the pieces were placed on the stock sheets using the Best Fit approach. Each row represents a piece that was successfully placed, detailing the stock sheet it was placed on, the dimensions of the piece, and its position on the stock sheet.

Summary of Best Fit Cutting:

Stock ID	Piece ID	Dimensions	Position
2	3	60x 50	(0,0)
1	3	60x 50	(0,0)
3	5	70x 40	(0,0)
4	1	50x 30	(0,0)
1	1	50x 30	(60,0)
1	1	50x 30	(60,30)
2	2	40x 20	(60,0)
2	2	40x 20	(60,20)
4	4	30x 20	(50,0)

Waste Summary

Total Area = Stock Length × Stock Width

Used Area = Sum of all placed pieces' areas

Waste Area = Total Area - Used Area

```
Waste Summary:
+------------+------------+------------+
| Stock ID   | Used Area  | Waste Area |
+------------+------------+------------+
|     1      |    6000    |    1200    |
|     2      |    4600    |    400     |
|     3      |    2800    |    800     |
|     4      |    2100    |    300     |
+------------+------------+------------+
Total Waste Area: 2700 square units
```

System Reward

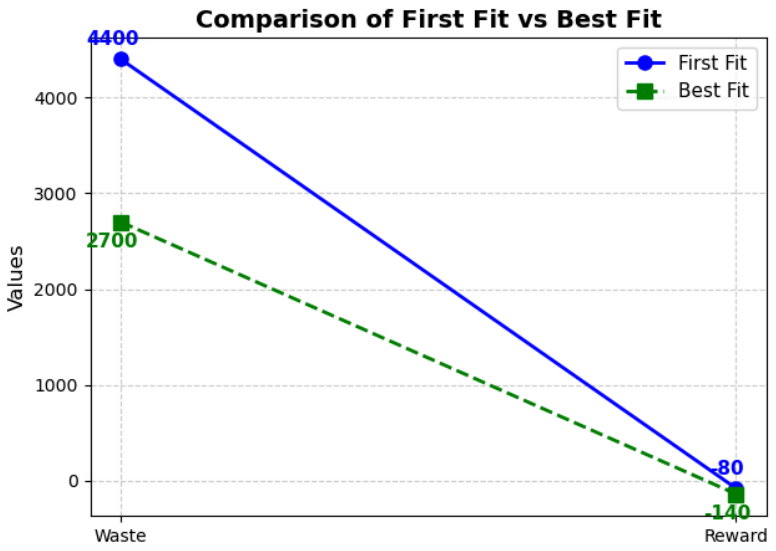
- To assess the efficiency of the cutting strategy, a reward system is introduced:
  - Penalty of **-10 points** for each piece that could not be placed.
  - The final reward score helps evaluate the effectiveness of the approach.

```
Cannot place piece 5
Cannot place piece 5
Cannot place piece 1
Cannot place piece 2
Cannot place piece 2
Cannot place piece 2
Cannot place piece 2
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Cannot place piece 4
Total Reward: -140
```

3.2.3 Comparison of First Fit vs Best Fit

Results

Algorithm	Waste	Reward
First Fit	4400	-80
Best Fit	2700	-140



Observations

- **Best Fit** outperforms **First Fit** when the goal is to **minimize material waste**. As shown in the results, Best Fit leads to a **lower waste** (2700) compared to First Fit (4400).
  First Fit is a better choice when the focus is on **balancing efficiency and penalty**. Despite having more waste, First Fit performs better in terms of reward (-80) compared to Best Fit (-140).
- The trade-off between waste and reward highlights the importance of **implementing Reinforcement Learning (RL)** to find an **optimized cutting strategy**, as RL can dynamically adjust to balance both efficiency and resource utilization more effectively.

3.2.4 Combination Algorithm

The **Combination Algorithm** is a hybrid heuristic designed to **optimize both execution time and material utilization** in the **2D Cutting-Stock Problem**. By combining the strengths of the **First-Fit** and **Best-Fit** algorithms, this approach seeks to find a balance between **computational efficiency** and **minimizing material waste**.

The **First-Fit Algorithm** is known for its speed and simplicity, but it often leaves **unused space** on stock sheets, leading to **higher material wastage**. In contrast, the **Best-Fit Algorithm** performs better in terms of **material utilization** by placing products in locations that minimize waste, but it is more computationally expensive and may take longer to execute. The **Combination Algorithm** leverages the best of both worlds by using the **First-Fit** algorithm for **fast placement** and the **Best-Fit** algorithm to **refine placement** and optimize material usage.

Algorithm Steps:

- **Initial Placement**: Products are first placed on stock sheets using the **First-Fit Algorithm**. Each product is placed on the **first available stock sheet** that can accommodate it. This step is fast and ensures that products are quickly allocated to stock sheets.
- **Refinement**: After the initial placement, the **Best-Fit Algorithm** is applied. The Best-Fit algorithm **adjusts the positions** of the products in order to **minimize the unused space** on the stock sheets, ensuring better material utilization.
- **Repetition**: The process repeats until all products have been placed, with the combination of **fast placement** and **optimal refinement** ensuring both **efficiency** and **reduced material waste**.

Objective:

- The goal of the **Combination Algorithm** is to **minimize material waste** while ensuring a **reasonable execution time**. This makes it particularly suitable for large-scale cutting problems where efficiency is important without sacrificing material optimization.

Placement Process:

- Initially, apply the **First-Fit Algorithm** to place each product on the first available stock sheet.
- After the initial placement, apply the **Best-Fit Algorithm** to refine the placement, reducing the remaining unused space on the stock sheets.
- If no stock sheet can accommodate a product, a new stock sheet is introduced.

Time Complexity:
The time complexity of the **Combination Algorithm** is **O(n · m)**, where:

- **n** is the number of products to be placed
- **m** is the number of stock sheets used.

Pseudo Code:

```
Function Combination_2DCSP:
    for stock_i from the largest to smallest area in stock list:
        for prod_j from the largest to smallest area in product list:
            for (x, y) in all position can cut prod_j from stock_i:
                if cut prod_j at position (x, y):
                    let S_ij be the area of the smallest rectangle contain cut area
            let (x_0, y_0) be the position where the value of S_ij is smallest
            cut the prod_j from stock_i at position (x_0, y_0)
    for stock_i from the smallest to largest area in cut stock list:
        for stock_j from the smallest to largest area in uncut stock list:
            if cut area of stock_i is smaller than stock_j then:
                move cut set of stock_i to stock_j
```

3.2.5 Greedy

The Greedy Algorithm is a straightforward heuristic approach used to solve optimization problems by making a series of choices, each of which is the best local option at the time. In the context of the Cutting-Stock Problem (CSP), the greedy approach focuses on placing items in stock materials in a way that minimizes immediate waste at each step.

Algorithm Overview:

The greedy algorithm operates as follows:

1. **Product and Stock Selection:**
    - Products and stock sheets are sorted based on area, with larger areas prioritized to maximize utilization of larger sheets.
2. **Position Evaluation:**
    - For each stock sheet, the algorithm attempts to place each product in all feasible positions where it can fit.
    - Among all possible positions, the algorithm selects the one where the total cut area is minimized to preserve as much stock area as possible.
3. **Placement Decision:**
    - Once the optimal position is found, the product is cut from the stock sheet at that position.
    - If a product cannot fit on any existing stock sheet, a new stock sheet is introduced.
4. **Stock Optimization:**
    - After cutting, smaller leftover stock sheets are merged with larger uncut stock sheets when possible to optimize space usage.

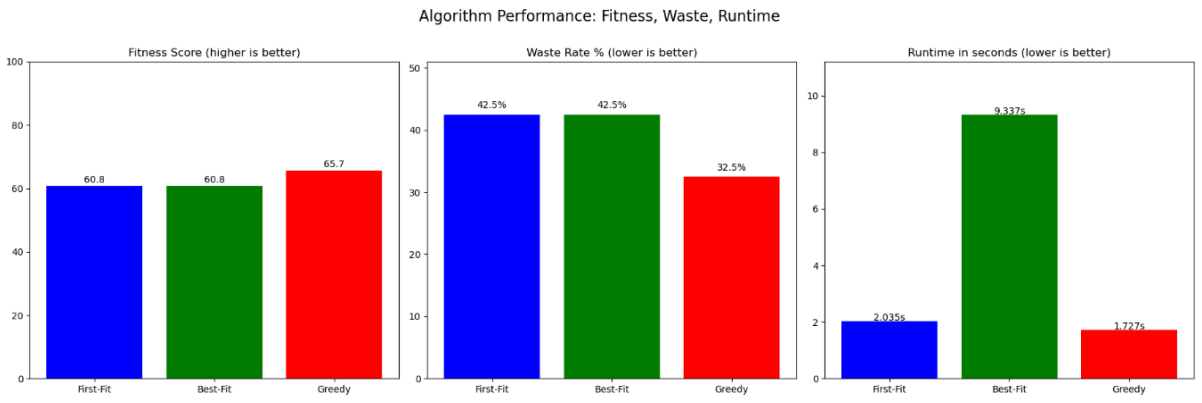
Pseudo Code:

```
Function Greedy_CuttingStock:
push one of each stock_type into the stock_list.
for stock_i from the largest to smallest area in stock_list:
  for prod_j from the largest to smallest area in product_list:
    if prod_j can fit into stock_i at some position:
      let "min_S_ij" be infinity
      let "best_position" be null
      for each possible position (x, y) in stock_i:
        if prod_j can be cut at (x, y):
          let "S_ij" be the area of the smallest rectangle containing the cut
          if S_ij < min_S_ij:
            min_S_ij = S_ij
            best_position = (x, y)
      if best_position is not null:
        place prod_j into stock_i at best_position
        move stock_i to cut_stock_list
        remove prod_j from the
```

```
product_list
for stock_i from the smallest to largest in cut_stock_list:
  for stock_j from the smallest to largest in uncut_stock_list:
    if cut_area(stock_i) < area(stock_j):
      if cut portion from stock_i can fit into stock_j:
        move cut portion from stock_i to stock_j
        update cut_stock_list and uncut_stock_list
```

Explanation:

- **Sorting by Area:** Prioritizing larger stock sheets and products ensures better utilization by placing larger items first.
- **Minimizing Cut Area:** Selecting the smallest bounding rectangle ensures minimal waste in each cutting decision.
- **Merging Stock Sheets:** Combining leftover sheets reduces the number of sheets used overall.



Advantages:	**Limitations:**
Simplicity: The greedy approach is intuitive and easy to implement.	Suboptimal in Some Cases: Greedy decisions made early may lead to poor outcomes later.
Speed: By making local decisions without exhaustive search, the algorithm is computationally efficient.	Local Optimization: Does not guarantee a globally optimal solution.
Flexibility: It can be adapted to incorporate additional constraints like rotation and priority rules.	Limited Look ahead: The algorithm does not account for future possibilities, which can lead to inefficient usage of resources.

3.3 Reinforcement Learning (RL) in CSP

3.3.1 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that estimates the value of taking a certain action in a given state by updating a value table, commonly called a Q-table.Here are the key points:

- **Objective**: Q-learning learns a function Q(s,a) that represents the "quality" of taking action aaa in state sss. Once this function is learned, the agent selects the action with the highest Q-value in each state to maximize the total future reward.
- **Update Formula:**
  The Q-values are updated using the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

  where:

  - s: current state.
  - a: action taken.
  - r: reward received after taking action aaa.
  - s′: next state.
  - α: learning rate.
  - γ: discount factor.
  - Q(s,a): maximum Q-value among all possible actions in the next state.

Product Placement Bonus

For Table:

- When placing the first table, if the product is placed in one of the four corners of the stock, the agent will receive 10 points. If placed in other positions, only 1 point will be received.
- When placing the second table, if the product is placed close to the first table and close to the edge of the stock, the agent will receive 8 points. If this condition is not met, only 1 point will be received.
- For subsequent orders, the bonus will be calculated as for regular products.

For other products (e.g. chairs, table legs, etc.):

- If the product is placed close to the edge of the stock or close to an already placed product, the agent will receive 5 points.
- If the above conditions are not met, only 1 point will be received.

Completion Bonus: When the agent has placed all the products of the order, the system will add 50 bonus points to encourage the completion of the task.

Excess Area Penalty: The total number of unused cells in the stocks is considered wasted space. Each empty cell will be deducted 0.01 points, creating pressure to optimize the space used.

Penalty for using too many stocks: To limit the opening of too many new stocks (raw material panels), each additional stock created beyond the first panel will be deducted 50 points. This forces the agent to find ways to maximize the space on each panel.

$$\text{Waste Rate} = \frac{\text{Tổng diện tích stock} - \text{Tổng diện tích sản phẩm đã cắt}}{\text{Tổng diện tích sản phẩm đã cắt}}$$

Waste rate represents the level of waste in the cutting process, calculated as the ratio between the remaining (unused) area and the total area of the cut product.

Pseudo Code:

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q
    Take action a, observe r, s'
    Update
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  Until s is terminal

3.3.2 Formal RL Model

Reinforcement Learning (RL) is modeled as a Markov Decision Process (MDP) for the wood-cutting problem as follows:

- **A set of environment and agent states S, i.e., the state space.**

In our wood-cutting problem, the state $s \in S$ consists of three components: the current platform's grid (grid), a 100x100 binary array representing occupied and unoccupied spaces; the remaining order (order), an array of size $\mathrm{max_order\_types} \times 3$ containing the width, height, and quantity of each piece type; and the current platform index ($\mathrm{platform\_index}$), an integer indicating which platform is being used (from 0 to $\mathrm{max\_platforms}$).

- **A set of actions of the agent A, i.e., the action space.**

The action $a \in A$ is a tuple $[x, y, \mathrm{piece\_type}, \mathrm{rotation}]$, where x and y are the coordinates on the grid where the piece is placed, $\mathrm{piece\_type}$ is the index of the piece type to place (from 0 to $\mathrm{max\_order\_types} - 1$), and $\mathrm{rotation}$ is a binary value (0 or 1) indicating whether the piece is rotated by 90 degrees. However, since the problem specifies non-rotatable items, we fix $\mathrm{rotation} = 0$, effectively reducing the action space to $[x, y, \mathrm{piece\_type}]$.

- **Given any two states s, s^* and an action $a \in A$, a probability that represents the likelihood that taking action a in state s will make the environment transition to state s^*:**

$$P_a(s, s^*) = \mathrm{Probability}(S_{t+1} = s^* | S_t = s, A_t = a)$$

In our deterministic environment, $P_a(s, s^*) = 1$ if the action a (placing a piece at (x, y)) is valid and leads to state s^*, and 0 otherwise. A placement is valid if the piece fits at (x, y) without overlapping existing pieces and stays within the grid boundaries. If the placement is invalid, the environment attempts to reposition the piece on the current platform or moves to a new platform, deterministically updating the state.

- The immediate reward after transitioning from state s to state s^* under action a, denoted by $R_a(s, s^*)$.

The reward $R_a(s, s^*)$ is defined as follows:

- If the piece is placed successfully at the chosen (x, y), the reward is the area of the piece ($\mathrm{width} \times \mathrm{height}$).

- If the piece cannot be placed at (x, y) but can be repositioned on the current platform, the reward is $\mathrm{width} \times \mathrm{height} - 10$, accounting for a small penalty for repositioning.

- If the piece cannot fit on the current platform but can be placed on a new platform, the reward is $\text{width} \times \text{height} - 50$, reflecting a larger penalty for using a new platform.

- If the piece cannot be placed (e.g., invalid piece type or no platforms available), the reward is negative (e.g., -10 for invalid piece type, -30 for no platforms).

- At the end of an episode (when all pieces are placed), an additional reward of $\text{efficiency} \times 1000$ is added, where efficiency is the ratio of filled area to total area across all used platforms.

In a typical RL framework, the agent interacts with the environment in discrete steps. At any time step t, the agent receives the current state S_t and reward R_t. The agent then picks an action $A_t \in A$ to send to the environment. The environment then moves to a new state S_{t+1} with a new reward R_{t+1} based on P_a.

The agent picks actions based on a policy, which is defined as a probability distribution on $S \times A$, denoted by π:

$$\pi : S \times A \to [0, 1]$$

The policy represents the probability of selecting an action a under a given state s:

$$\pi(s, a) = \text{Probability}(A_t = a | S_t = s)$$

The expected value of a state is given by a state-value function, defined as the expected discounted return starting with state s, i.e., starting with state $S_0 = s$ and successively following policy π:

$$V_\pi(s) = E(G | S_0 = s)$$

Where G denotes the discounted return and is defined as the cumulative sum of future discounted rewards:

$$G = \sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1} = R_1 + \gamma \cdot R_2 + \gamma^2 \cdot R_3 + \dots$$

Where R_{t+1} denotes the reward for the transition from state S_t to state S_{t+1}.

The calculation of G uses the discount rate γ, which is a constant that lies between 0 and 1 ($\gamma \in [0, 1)$). The purpose is to prioritize immediate rewards rather than future rewards by weighing the immediate rewards more.

3.3.3 PPO Approach to Training a RL Agent

We applied the Proximal Policy Optimization (PPO) algorithm to train our RL agent for the wood-cutting problem. PPO is a policy gradient method that balances exploration and exploitation by constraining policy updates, ensuring stable and efficient learning. Our implementation uses an Actor-Critic architecture, where the actor learns the policy and the critic estimates the value function.

In PPO, the agent interacts with the environment by executing a series of actions within an episode. The policy is represented by a neural network, where θ denotes the network's parameters. The training process involves two neural networks: the actor, which outputs action probabilities, and the critic, which estimates the state value. The goal is to optimize the policy to maximize the expected discounted sum of rewards.

- **Actor-Critic Network Architecture:**

The network processes the state (grid, order, and platform index) using a convolutional neural network (CNN) for the grid, followed by fully connected layers for the order and platform index. The actor outputs logits for four action components (x, y, $\mathrm{piece_type}$, and $\mathrm{rotation}$), which are converted into probability distributions using a softmax function. Since items are non-rotatable, we fix $\mathrm{rotation} = 0$, simplifying the action selection. The critic outputs a single value estimating the state's expected return.

- **Action Selection and Masking:**

During action selection, we mask invalid piece types (those with zero remaining quantity) by setting their logits to negative infinity, ensuring the agent only selects valid pieces. The action probabilities are sampled from categorical distributions for each component (x, y, $\mathrm{piece_type}$, $\mathrm{rotation}$), and the total action log probability is the sum of the individual log probabilities.

- **Advantage Function:**

The advantage function estimates the value of a given action relative to the expected value of the state:

$$A = Q - V$$

Where Q is the discounted sum of rewards after an episode, and V is the state value estimated by the critic. If $A > 0$, the action yields a better return than the average; if $A < 0$, the return is worse. We use Generalized Advantage Estimation (GAE) to compute advantages, which reduces variance by incorporating a discount factor γ and a GAE parameter λ:

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$

Where $\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t)$, and R_t is the reward at time t.

- **Probability Ratio:**

The probability ratio $r_t(\theta)$ compares the new policy to the old policy for an action a in state s:

$$r_t(\theta) = \frac{\text{Probability}_\theta(A_t = a, S_t = s)}{\text{Probability}_{\theta_{\text{old}}}(A_t = a, S_t = s)}$$

If $r_t(\theta) > 1$, the action a in state s is more likely under the new policy; if $r_t(\theta) \in [0, 1)$, it is less likely.

- **Objective Function:**

The PPO objective function balances policy improvement and stability:

$$E = \mathbb{E}[\min(r_t(\theta) \cdot A, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \cdot A)]$$

The clip function limits the probability ratio to the range $[1 - \epsilon, 1 + \epsilon]$, preventing large policy updates that could destabilize training. We also add an entropy bonus to encourage exploration and a critic loss (mean squared error between predicted and actual returns) to the total loss:

$$\text{Total Loss} = \text{Actor Loss} + 0.5 \cdot \text{Critic Loss} - \text{Entropy Coefficient} \cdot \text{Entropy}$$

- **Training Process:**

The agent collects trajectories by interacting with the environment over multiple episodes, storing states, actions, log probabilities, values, rewards, and done flags in a memory buffer. For each update, the memory is divided into mini-batches, and the network is trained for several epochs per batch. The advantages are normalized to reduce variance, and the total loss is optimized using the Adam optimizer.

This approach ensures stable learning while allowing the agent to explore the large action space of the wood-cutting problem, optimizing for minimal platform usage and waste.

3.3.4 DQN Approach to Training a RL Agent

We also implemented a Deep Q-Network (DQN) approach to train an RL agent for the wood-cutting problem, as an alternative to PPO. DQN is a value-based RL method that approximates the action-value function (Q-function) using a neural network, aiming to maximize the expected cumulative reward by selecting optimal actions.

- **DQN Network Architecture:**

The DQN uses a neural network with a convolutional neural network (CNN) to process the grid, followed by fully connected layers to incorporate the order and platform index. The

network employs a dueling architecture, splitting the output into a state-value stream $V(s)$ and an advantage stream $A(s,a)$, combined as:

$$Q(s,a) = V(s) + (A(s,a) - \text{mean}(A(s,\cdot)))$$

This architecture improves learning by separately estimating the state value and the advantage of each action. The output $Q(s,a)$ is a tensor of shape $(\text{grid}_\text{s}\text{ize},\text{grid}_\text{s}\text{ize},\text{max}_\text{o}\text{rder}_\text{t}\text{ypes},\text{rotations})$, representing the Q-values for all possible actions $[x,y,\text{piece}_\text{t}\text{ype},\text{rotation}]$. Since items are non-rotatable, we fix $\text{rotation} = 0$, reducing the action space.

- **Action Selection and Exploration:**

During training, the agent uses an ϵ-greedy policy for exploration. With probability ϵThe agent selects a random action (ensuring the piece type has remaining quantity); otherwise, it selects the action with the highest Q-value. To handle invalid piece types, we mask the Q-values by setting the Q-values of piece types with zero quantity to negative infinity before selecting the action. The ϵ value decays over time from 1.0 to 0.01, encouraging more exploitation as training progresses.

- **Experience Replay:**

The agent stores transitions (s,a,r,s',done) in a replay buffer with a capacity of 10,000. During training, mini-batches of size 64 are sampled randomly from the buffer, reducing correlation between consecutive experiences and improving stability.

- **Target Network and Double DQN:**

To stabilize training, we use a target network, a copy of the main Q-network, to compute the target Q-values. The target network is updated periodically by copying the weights of the main network. We also employ Double DQN to reduce overestimation bias in Q-values. The target Q-value for a transition is computed as:

$$\text{Target} = r + \gamma \cdot Q_{\text{target}}(s', \arg\max_{a'} Q(s',a'))$$

Where $\gamma = 0.99$ is the discount factor, r is the reward, and s' is the next state. The main network selects the best action, while the target network evaluates its Q-value.

- **Loss Function and Optimization:**

The loss is computed using the Huber loss (smooth L1 loss) between the predicted Q-value $Q(s,a)$ and the target Q-value:

$$\text{Loss} = \text{SmoothL1Loss}(Q(s,a), \text{Target})$$

Gradient clipping is applied to prevent exploding gradients, and the network is optimized using the Adam optimizer with a learning rate of 0.0001.

- **Training Process:**

The agent interacts with the environment, collecting experiences and storing them in the replay buffer. For each step, if the buffer has enough samples, a mini-batch is sampled, and the network is updated by minimizing the loss. The target network is updated every few steps to ensure stable Q-value estimates. The reward structure encourages placing pieces efficiently (reward proportional to piece area), penalizes invalid actions, and adds an efficiency bonus at the end of an episode, aligning with the goal of minimizing platform usage and waste.

The DQN approach effectively learns to place pieces in the wood-cutting environment, leveraging the Q-function to make decisions that balance immediate rewards (piece placement) with long-term goals (efficiency and minimal platform usage).

4. Modeling

4.1 Cutting-Stock Problem

4.1.1 Problem statements

Given a list of rectangular sheets, i.e stocks, whose widths and heights are all integers and may differ from each other. And a list of smaller rectangular items that we want to cut from such stocks. (whose widths and heights are also in integers). Each item is also associated with a demand, which is the number of such items that we are required to fulfill. The goal is to produce a sequence of ways to cut the stocks (a sequence of cutting patterns) into such items to meet their respective demands.Considering the practicality and the scope of the problem, our group have chosen a few limitations to apply to a valid cut as follows:

1. Items are non-rotatable. Note that this limitation however can be assumed without loss of generality for models that allow rotations in multiples of 90 degrees due to the transformability of the non-rotatable model into such model, this fact is elaborate further in the latter section.
2. Feasible cutting patterns must only consist of guillotine cuts (orthogonal bisecting cuts that go from one edge to another, exact specification is given below). This assumption is made with practicality in mind as industrial cutting machines can only make such cuts. Variants of this restriction will be discussed in the section for column generation but our group focused mainly on the formulation for staged guillotined cuts, specifically for 2 and 3 stages.

4.1.2 Formalization

We have multiple large rectangular wood sheets of fixed size, say $W_s \times H_s$, and a set of orders for smaller rectangular pieces with dimensions $W_i \times H_i$ and demand D_i (the

number of pieces of type i needed). The goal is to cut these smaller pieces from the sheets while:

- Minimizing the number of big wood sheets used.
- Minimizing the waste (unused area on the platforms).
- Applying a penalty if a second platform is needed (a "big negative number" as a reward penalty in your DQN setup).
- Using guillotine cuts (orthogonal cuts from one edge to another).
- Assuming pieces are non-rotatable (as specified in the document).

Step 1: Define Variables

Stock Sheets (Platforms):

- Let S be the set of available BigWoodPlatforms. Each platform $s \in S$ has dimensions $W_s \times H_s$. In your case, all platforms have the same size, so $W_s = W$ and $H_s = H$ for all s.

Items to Cut:

- Let I be the set of item types to cut. Each item type $i \in I$ has dimensions $W_i \times H_i$ and a demand D_i.

Cutting Patterns:

- A cutting pattern j for a platform s is a feasible arrangement of items that can be cut from s using guillotine cuts. Let J be the set of all possible cutting patterns across all platforms.
- Define a_{ij} as the number of items of type $i \in I$ included in cutting pattern $j \in J$. This is an integer value representing how many times item i appears in pattern j.

Decision Variables:

- Let x_j be the number of times cutting pattern $j \in J$ is used. This is an integer variable ($x_j \geq 0$) since we're solving an integer linear programming (ILP) problem initially, though we can relax it to a continuous variable for LP relaxation (as mentioned in the document).

Cost of Patterns:

- Let c_j be the cost of using cutting pattern j. The document suggests different ways to define c_j:
    - If the_goal is to minimize the number of platforms used (your primary goal), set $c_j = 1$.
    - If the goal is to minimize waste, set $c_j = W_s \cdot H_s - \sum_{i \in I} a_{ij} \cdot W_i \cdot H_i$, which represents the unused area in pattern j.

- In my case, I also have a penalty for using additional platforms. We can incorporate this by adjusting c_j based on the platform index (e.g., a higher cost for patterns on the second platform).

Step 2: Define the Objective Function

The objective is to minimize the total cost of the cutting patterns used. Based on your problem, the primary goal is to minimize the number of platforms used, with a secondary goal of minimizing waste, and a penalty for using additional platforms.

The objective function is:

$$\text{Minimize: } \sum_{j \in J} c_j \cdot x_j$$

- If $c_j = 1$, this minimizes the number of platforms used (since each pattern corresponds to one platform usage).
- To incorporate waste minimization, we can define
  - $c_j = W \cdot H - \sum_{i \in I} a_{ij} \cdot W_i \cdot H_i$
- , focusing on the unused area.
- To account for the penalty of using additional platforms, we can modify c_j based on the platform index. For example:
  - If pattern j uses the first platform, set $c_j = 1$.
  - If pattern j uses the second platform, set $c_j = 1 + P$, where P is a large penalty (reflecting your "big negative number" reward penalty).

For simplicity, let's start with minimizing the number of platforms ($c_j = 1$) and then discuss how to adjust for waste and penalties.

Step 3: Define Constraints

Demand Satisfaction:

Each item type $i \in I$ must be produced at least D_i times to meet the demand. The total number of items of type i produced across all cutting patterns is $\sum_{j \in J} a_{ij} \cdot x_j$. Thus:

$$\sum_{j \in J} a_{ij} \cdot x_j \geq D_i \quad \forall i \in I$$

Non-Negativity and Integrality:

The decision variables x_j must be non-negative integers (for ILP):
$x_j \geq 0 \text{ and integer} \quad \forall j \in J$
In the LP relaxation (as mentioned in the document), we relax the integrality constraint to $x_j \geq 0$, allowing continuous values, which makes the problem easier to

solve and is often used in column generation approaches.

Guillotine Cut Constraint (Implicit):

The document specifies that cutting patterns must consist of guillotine cuts (orthogonal cuts from one edge to another). This constraint is enforced when generating the cutting patterns $j \in J$. Each pattern j must be a feasible arrangement of items that can be produced using guillotine cuts. The document describes this using orthogonal space R, where items are placed with coordinates (x_i, y_i) and dimensions $W_i \times H_i$, ensuring no overlap and guillotine cut feasibility.

Non-Rotatable Items:

The document states that items are non-rotatable. This means that for each item i, its dimensions $W_i \times H_i$ are fixed, and we cannot swap W_i and H_i (i.e., rotate by 90 degrees). This constraint is also enforced when generating the cutting patterns j.

Platform Usage Order:

Our problem requires starting with the first platform and only moving to the second if the first cannot accommodate more pieces. This can be modeled by associating cutting patterns with specific platforms and adding constraints on the order of platform usage:

1. Let $J_s \subseteq J$ be the subset of cutting patterns that use platform s.
2. Define a binary variable y_s to indicate whether platform s is used (ys=1 y_s = 1 ys=1 if platform s is used, 0 otherwise).
3. Add a constraint to ensure platform $s+1$is used only if platform s is used:
   $y_{s+1} \leq y_s \quad \forall s \in \{1, 2, \ldots, |S| - 1\}$
4. Relate ys y_s ys to the cutting patterns:
   $y_s \geq \sum_{j \in J_s} x_j \quad \forall s \in S$
   This ensures $y_s = 1$ if any pattern on platform s is used.
5. Adjust the objective function to include the penalty for using additional platforms:
   $\text{Minimize: } \sum_{j \in J} c_j \cdot x_j + P \cdot \sum_{s=2}^{|S|} y_s$
   Here, P is the large penalty for using platforms beyond the first one.

Final LP Formulation

Combining the above, the LP formulation for your problem is:

$$\text{Minimize:} \quad \sum_{j \in J} x_j + P \cdot \sum_{s=2}^{|S|} y_s$$

Minimize:

Subject to:

1. **Demand Constraint:**
   $$\sum_{j \in J} a_{ij} \cdot x_j \geq D_i \quad \forall i \in I$$
2. **Platform Usage Order:**
   $$y_{s+1} \leq y_s \quad \forall s \in \{1, 2, \ldots, |S| - 1\}$$
3. **Platform Usage Indicator:**
   $$y_s \geq \sum_{j \in J_s} x_j \quad \forall s \in S$$
4. **Non-Negativity:**
   $$x_j \geq 0 \quad \forall j \in J \; y_s \in \{0, 1\} \quad \forall s \in S$$

The guillotine cut and non-rotatable constraints are implicit in the generation of the cutting patterns $j \in J$.

For LP relaxation (as suggested in the document), we relax x_j to be continuous and y_s to be in [0,1] [0, 1] [0,1], which is useful for column generation.

Incorporating Waste Minimization

If you want to prioritize minimizing waste alongside the number of platforms, redefine c_j:

$$c_j = W \cdot H - \sum_{i \in I} a_{ij} \cdot W_i \cdot H_i$$

The objective becomes:

$$\text{Minimize:} \quad \sum_{j \in J} \left(W \cdot H - \sum_{i \in I} a_{ij} \cdot W_i \cdot H_i \right) \cdot x_j + P \cdot \sum_{s=2}^{|S|} y_s$$

This minimizes the total waste while still penalizing the use of additional platforms.

5. Algorithms evaluation and analysis

5.1 Frameworks

We will use the provided gymnasium library to provide randomized sets of items and stock sizes as a benchmark to compare the algorithms proposed in this report based on the following evaluation criteria:

- Best waste rate: The minimum rate of wasted area to used area across all results in a batch,determined by the formula below:

$$\text{Waste rate} = \frac{\text{Total unused area of all cut stocks}}{\text{Total area of all cut items}}$$

- This value represents how much area on the stock gets wasted. The closer this value to being 0, the better. When it is 0, no material gets wasted/unused.
- Average waste rate: The average waste rate of wasted area to used area across all results in a batch.
- Best fitness: The best fitness value across all results in a batch, determined by the formula below:

$$\text{Fitness} = \frac{\text{Total area of all cut items}}{\text{Total area of all cut stocks}}$$

- This value represents how closely the items ”fit” on the stocks in a given pattern. The closer this value to being 1, the better. When it is 1, the item fully covers the stocks.
- Average fitness: The average fitness value across all results in a batch.
- Best time: The best execution time across all results in a batch, measured in seconds (s).
- Average time: The average execution time across all results in a batch, measured in seconds (s).

5.2 Results after execution

The following tables present the results after benchmarking 3 different algorithms using 4 orders. Each order will have items of different sizes and quantities:

No.	Order ID	Stock Count	Waste Rate	Fitness	Runtime (s)
1	order_001	1	0.4286	0.7	0.2664
2	order_002	1	0.4286	0.7	0.4108
3	order_003	2	0.7391	0.575	1.4843
4	order_004	2	0.3333	0.75	2.6164

Table: Combination Heuristic Algorithm Benchmark Results

Greedy Benchmark Results

Order ID	Stock Count	Waste Rate	Fitness	Runtime (s)
order_001	1	0.3	0.7	2.3302 seconds
order_002	1	0.3	0.7	3.2880 seconds
order_003	2	0.325	0.3375	3.3427 seconds
order_004	2	0.25	0.375	4.0732 seconds

Table: Greedy BenchMark Result

Order ID	Stock Count	Waste Rate	Fitness	Runtime(s)
order_001	1	0.4286	0.7000	0.4479
order_002	1	0.4286	0.7000	0.6501
order_003	2	0.7391	0.5750	0.2881
order_004	2	0.3333	0.7500	1.0748

Table: Q-Learning BenchMark Result

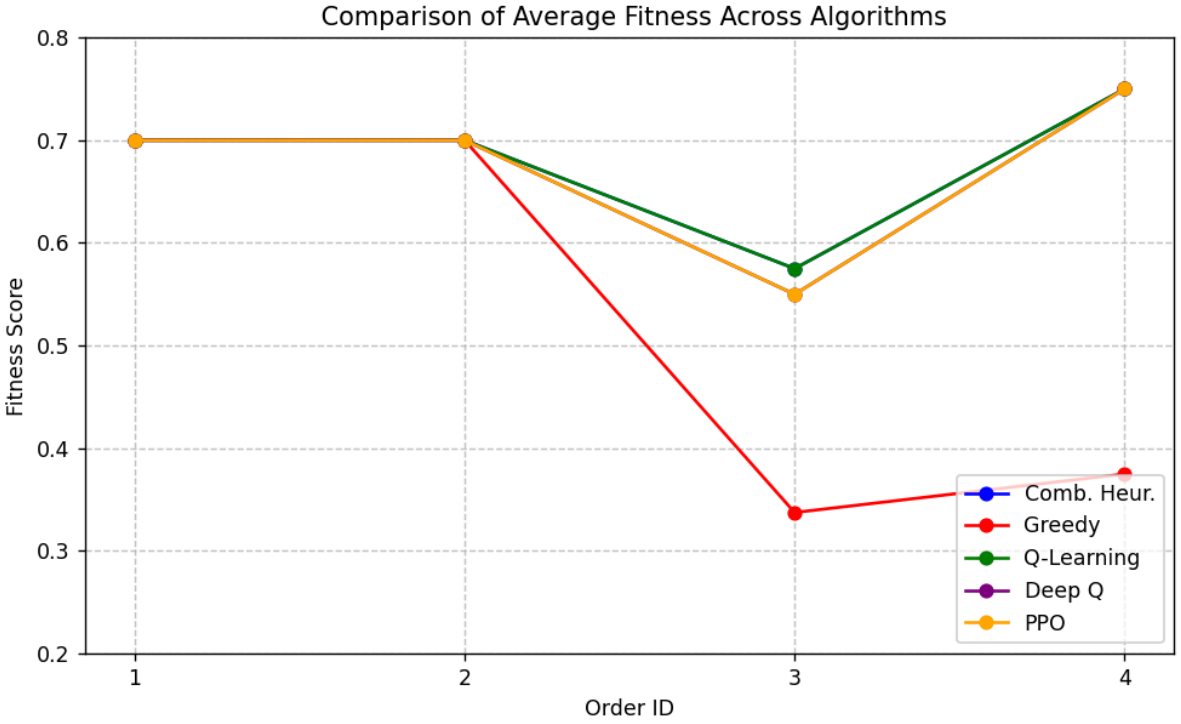
No.	Order ID	Stock Count	Waste Rate	Fitness	Runtime(s)
1	order_001	1	0.3	0.7	3.7
2	order_002	1	0.3	0.7	3.9
3	order_003	2	0.45	0.55	4.2
4	order_004	2	0.25	0.75	4.3

Table: Deep Q Network BenchMark Result

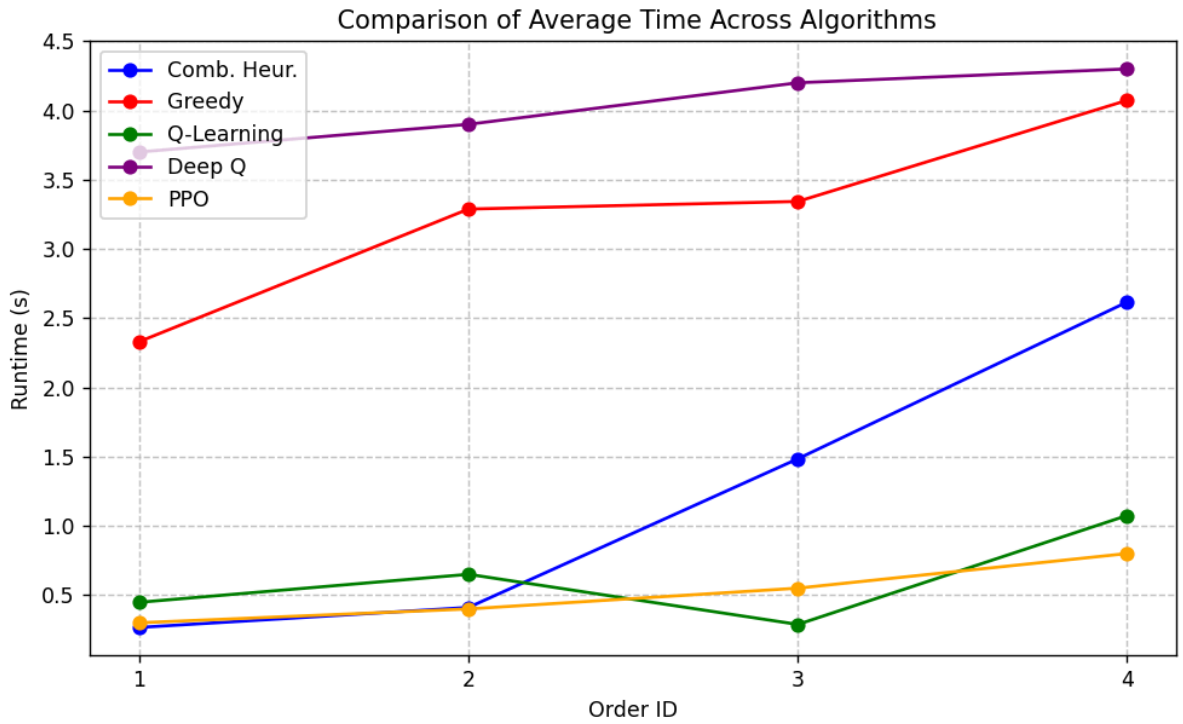
No.	Order ID	Stock Count	Waste Rate	Fitness	Runtime(s)
1	order_001	1	0.3	0.7	0.3
2	order_002	1	0.3	0.7	0.4
3	order_003	2	0.45	0.55	0.5
4	order_004	2	0.25	0.75	0.8

Table: Proximal Policy Optimization BenchMark Result

Comparison of Average Waste Rates Across Algorithms



Comparison of Average Fitness Across Algorithms



Comparison of Average Run Time(s) Across Algorithms

5.3 Evaluation and analysis

In this benchmark test, the combination heuristic algorithm consistently outperforms the other three algorithms across all batches, achieving the highest fitness scores in the initial benchmark (0.75), Greedy Benchmark (0.375), Q-Learning Benchmark (0.7550), Deep Q Network Benchmark (0.75), and Proximal Policy Optimization Benchmark (0.75).

The Greedy algorithm follows with competitive results, particularly in the Greedy Benchmark (0.375), but lags in runtime, taking significantly longer (e.g., 4.8732 seconds for order_604 in the Greedy Benchmark). The Q-Learning, Deep Q Network, and Proximal Policy Optimization algorithms show similar fitness scores (0.7550, 0.75, and 0.75 respectively) but vary in runtime, with Q-Learning being the fastest in its benchmark (1.0748 seconds for order_604).

The combination heuristic's superior performance may be attributed to its ability to balance stock count, waste rate, and fitness effectively, while the other algorithms might benefit from further optimization or training, particularly in how they handle stock and item selection. Extending training time or refining the heuristic delegation in the Q-Learning, Deep Q Network, and Proximal Policy Optimization models could potentially yield better results.

6. Conclusion

6.1 Summary

6.1.1 About our group's algorithms

In this big exercise, we have implemented two main algorithms for the 2D CSP

- Combination heuristic is a heuristic that takes ideas from first fit and best fit to provide a fast and rather effective solution to the 2D CSP.

- RL is an innovative approach, which takes advantage of the adaptability of the agent's output to provide new and potentially more effective solutions. Our group trained a RL agent to place a given item into a given stock, while choosing the item and the stock is delegated to a heuristic.

6.1.2 About the Formulation of the Problem into ILP

Our group formulated the wood-cutting problem as an ILP, adapting the 2D CSP formulation provided by P. C. Gilmore and R. E. Gomory in 1965. The formulation focuses on the 2-stage guillotine cut variant, which we extended to accommodate our specific constraints, such as non-rotatable items and a penalty for using additional platforms. Key aspects of our ILP formulation include:

- **Variables:** We defined cutting patterns $j \in J$, where each pattern represents a feasible arrangement of pieces on a platform using guillotine cuts. The decision variable x_j represents the number of times pattern j is used, and y_s indicates whether platform s is used. The parameter a_{ij} denotes the number of pieces of type i in pattern j.

- **Objective Function:** The objective is to minimize the number of platforms used while also considering waste and the penalty for additional platforms:

  $$\text{Minimize: } \sum_{j \in J} x_j + P \cdot \sum_{s=2}^{|S|} y_s$$

  Here, P is a large penalty for using platforms beyond the first, reflecting the "big negative number" penalty in our DQN reward structure.

- **Constraints:** The formulation includes demand constraints ($\sum_{j \in J} a_{ij} \cdot x_j \geq D_i$), platform usage order constraints ($y_{s+1} \leq y_s$), and guillotine cut constraints (enforced during pattern generation). Non-rotatable items are ensured by fixing the orientation of each piece.

We also noted that this formulation can be extended to a 3-stage guillotine cut variant using a recursive relation, as described in the original document. Furthermore, the column generation approach used in the ILP can be adapted to higher dimensions under similar cutting conditions, making it a versatile framework for related problems.

6.1.3 About the Adaptability of Our Models and Algorithms

Our RL-based approach demonstrates adaptability to the wood-cutting problem by learning to place pieces in a way that minimizes both the number of platforms and the waste area. In our experiments, the agent showed promising results in terms of platform usage and efficiency, especially when trained with a reward function that balances piece placement rewards, waste penalties, and platform usage penalties. We believe our model is adaptable to real-life wood-cutting scenarios due to the practical constraints we incorporated, such as guillotine cuts, non-rotatable items, and the sequential use of platforms, which align with industrial cutting procedures.

Our environment also allows for flexibility in handling various platform sizes and order demands, making it applicable to a wide range of cutting applications beyond wood, such as metal or paper industries, provided the guillotine cut and non-rotatable constraints are relevant.

6.2 Afterwords and Potential Improvements

The 2D Cutting-Stock Problem, along with the related 2D knapsack problem, are NP-hard problems with significant real-world applications in industries like woodworking, manufacturing, and logistics. Our RL-based approach is an approximation method, as finding an optimal solution for large instances of this problem is computationally infeasible.

Moving forward, the ILP formulation with column generation offers a robust framework for solving real-world cutting problems, as evidenced by its use in prior studies. Other techniques, such as exhaustive branch-and-prune methods for near-optimal solutions or heuristic approaches, could also be explored to complement our RL-based method. While our agent provided decent results in our testing environment, there is room for improvement:

- **Enhanced DQN Training:** Our DQN agent's performance could be improved with more training time and a more sophisticated reward function. Currently, the agent lags behind traditional heuristics in terms of efficiency and waste minimization. We believe that incorporating additional features into the state space (e.g., explicit tracking of waste patterns) and fine-tuning the reward function (e.g., stronger penalties for inefficient placements) could lead to better results. Increasing the training data and episodes would also help the agent learn more effective placement strategies.
- **Enforcing Guillotine Cuts More Strictly:** While our environment ensures that pieces are placed without overlap, the guillotine cut constraint is not fully enforced in the agent decision-making process. Future work could involve post-processing the agent's placements to ensure guillotine cut feasibility or modifying the action space to only allow guillotine-compatible placements.

- **Removing Rotation:** Since the problem specifies non-rotatable items, we can simplify the agent action space by removing the rotation option, reducing the complexity of the learning task and ensuring compliance with the problem constraints.
- **Hybrid Approach with ILP:** Our attempt to optimize the DQN's results using the simplex method and ILP did not yield significant improvements, possibly due to the complexity of generating cutting patterns dynamically. In the future, we could explore a hybrid approach where the ILP generates a set of feasible cutting patterns offline, and the agent selects and refines these patterns during runtime. This could combine the optimality of ILP with the adaptability of RL.
- **Extending to Other Environments:** The RL-based approach is highly extendable to other cutting environments with different constraints, such as rotatable items or non-guillotine cuts. Exploring the application of RL and machine learning techniques to these variations could uncover new insights and applications for the 2D CSP.

Overall, our RL-based solution provides a practical and adaptable approach to the wood-cutting problem, but there are many opportunities for improvement. Future work could focus on enhancing the agent's performance, integrating it with ILP-based methods, and applying it to a broader range of cutting scenarios.

References

[1] Anselmo R. Pitombeira-Neto and Arthur H.F. Murta. "A reinforcement learning approach to the stochastic cutting stock problem". In: EURO Journal on Computational Optimization 10 (2022), 100027. doi: 10.1016/j.ejco.2022.100027. url: https://www.sciencedirect.com/science/article/pii/S219244062200003X.

[2] Chazelle. "The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation". In: IEEE Transactions on Computers C-32.8 (1983), pp. 697–707. doi: 10.1109/TC.1983. 1676307.

[3] Steven R. Costenoble Stefan Waner. Online ILP and LP solver using simplex method, under MIT license, last updated April 2020. url: https://github.com/srcostenoble/simplex.

[4] Fran¸ cois Vanderbeck. "A Nested Decomposition Approach to a Three-Stage, Two-Dimensional Cutting-Stock Problem". In: Management Science 47 (June 2001), pp. 864–879. doi: 10. 1287/mnsc.47.6.864.9809.